Lightweight Superscalar Task Execution in Distributed Memory

Asim YarKhan\textsuperscript{1} and Jack Dongarra\textsuperscript{1,2,3}

\textsuperscript{1}Innovative Computing Lab, University of Tennessee, Knoxville, TN
\textsuperscript{2}Oak Ridge National Lab, Oak Ridge, TN
\textsuperscript{3}University of Manchester, Manchester, UK

MTAGS 2014
7th Workshop on Many-Task Computing on Clouds, Grids, and Supercomputers
New Orleans, Louisiana

Nov 16 2014
Parallel Programming is difficult (still, again, yet).
- Coding is via Pthreads, MPI, OpenMP, UPC, etc.
- User handles complexities of coding, scheduling, execution, etc.

Efficient and scalable programming is hard
- Often get undesired synchronization points.
- Fork-join wastes cores and reduces performance.
- We need to access more of provided parallelism.
  - Larger multicore architectures
  - More inactive cores = more waste
Productivity, Efficiency, Scalability

- **Productivity** in Programming
  - Have a simple, serial API for programming.
  - Runtime environment handles all the details.

- **Efficiency** and **Scalability**
  - Tasks have data dependencies.
  - Tasks can execute as soon as data is ready (async).
  - This results in a task-DAG (directed acyclic graph).
  - Nodes are tasks; edges are data dependencies
  - Uses available cores in shared memory.
  - Transfers data as required in distributed memory
Related Projects

- **PaRSEC [UTK]**: Framework for distributed memory task execution. *Requires specialized parameterized compact task graph description*; parameterized task graphs are hard to express. Very high performance is achievable. Implements DPLASMA.

- **SMPss [Barcelona]**: Shared memory. Compiler-pragmas based, runtime-system with data locality and task-stealing, *emphasis on data replication*. MPI available via explicit wrappers.

- **StarPU [INRIA]**: Shared and distributed memory. Library API based, *emphasis on heterogeneous scheduling (GPUs)*, smart data management, - similar to this work.

- **Others**: Charm++, Jade, Cilk, OpenMP, SuperMatrix, FLAME, ScaLAPACK, ...
Driving Applications: Tile Linear Algebra Algorithms

- **Block algorithms**
  - Standard linear algebra libraries (LAPACK, ScaLAPACK) gain parallelism from BLAS-3 interspersed with less parallel operations.
  - Execution is fork-join (or block synchronous parallel).

- **Tile algorithms**
  - Rewrite algorithms as tasks acting on data tiles.
  - Tasks using data $\implies$ data dependencies $\implies$ DAG
  - Want to execute DAGs asynchronously and in parallel $\implies$ runtime.
  - QUeuing and Runtime for Kernels for Distributed Memory
Tile QR Factorization Algorithm

\[
\text{for } k = 0 \ldots \text{TILES}-1 \\
\text{geqrt}( A_{kk}^r, T_{kk}^w ) \\
\text{for } n = k+1 \ldots \text{TILES}-1 \\
\text{unmqr}( A_{kk-low}^r, T_{kk}^r, A_{kn}^w ) \\
\text{for } m = k+1 \ldots \text{TILES}-1 \\
\text{tsqrt}( A_{kk-up}^r, A_{mk}^r, T_{mk}^w ) \\
\text{for } n = k+1 \ldots \text{TILES}-1 \\
\text{tsmqr}( A_{mk}^r, T_{mk}^r, A_{kn}^w, A_{mn}^w )
\]

List of tasks as they are generated by the loops
Tile QR Factorization: Data Dependencies

Data dependencies from the first five tasks in the QR factorization

\[ A_{00} : F_{0}^{rw} : F_{1}^{r} : F_{2}^{r} : F_{3}^{rw} \]

\[ A_{01} : F_{1}^{rw} : F_{4}^{rw} \]

\[ A_{02} : F_{2}^{rw} : F_{5}^{rw} \]

\[ A_{10} : F_{3}^{rw} : F_{4}^{r} : F_{5}^{r} \]

\[ A_{11} : F_{4}^{rw} \]

\[ A_{12} : F_{5}^{rw} \]

\[ A_{20} : \]

\[ A_{21} : \]

\[ A_{22} : \]
First step in execution - Run task (function) $F_0$.

$F_0$ geqrt ( $A^{rw}_{00}$, $T^w_{00}$ )

$F_1$ unmqr ( $A^r_{00}$, $T^r_{00}$, $A^{rw}_{01}$ )

$F_2$ unmqr ( $A^r_{00}$, $T^r_{00}$, $A^{rw}_{02}$ )

$F_3$ tsqrt ( $A^{rw}_{00}$, $A^{rw}_{10}$, $T^w_{10}$ )

$F_4$ tsmqr ( $A^{rw}_{01}$, $A^{rw}_{11}$, $A^r_{10}$, $T^r_{10}$ )

$F_5$ tsmqr ( $A^{rw}_{02}$, $A^{rw}_{12}$, $A^r_{10}$, $T^r_{10}$ )

Second step in execution - Remove $F_0$; Now $F_1$ and $F_2$ are ready.

$A^{00}$: $F_0^{rw}$ : $F_1^r$ : $F_2^r$ : $F_3^{rw}$

$A^{01}$: $F_1^{rw}$ : $F_4^{rw}$

$A^{02}$: $F_2^{rw}$ : $F_5^{rw}$

$A^{10}$: $F_3^{rw}$ : $F_4^r$ : $F_5^r$

$A^{11}$: $F_4^r$

$A^{12}$: $F_5^r$

$T^{00}$: $F_0^w$ : $F_1^r$ : $F_2^r$

$T^{10}$: $F_3^w$ : $F_4^r$ : $F_5^r$
QUARK-D API and Runtime

- **QUARK-D**
  - QUeuing and Runtime for Kernels in Distributed Memory

- Simple serial task insertion interface.
  
  ```c
  QUARKD_Insert_Task( quark, *function, *taskflags,
      a_flags, size_a, *a, a_home_process, a_key,
      b_flags, size_b, *b, b_home_process, b_key,
      ... , 0 );
  ```

- Manage the distributed details for the user.
  - Scheduling tasks (where should tasks run)
  - Data dependencies and movement (local and remote).
  - Transparent communication.
  - No global knowledge or coordination required.
Productivity: QUARK-D QR Implementation

The code matches the pseudo-code

```c
#define A(m, n) ADDR(A), HOME(m, n), KEY(A, m, n)
#define T(m, n) ADDR(T), HOME(m, n), KEY(T, m, n)

void plasma_pdgeqrf(A, T, , ) {
    for (k = 0; k < M; k++) {
        TASK_dgeqrt(quark,, A(k,k), T(k,k));
        for (n = k+1; n < N; n++)
            TASK_dormqr(quark,, A(k,k), T(k,k), A(k,n));
        for (m = k+1; m < M; m++) {
            TASK_dtsqrt(quark,, A(k,k), A(m,k), T(m,k));
            for (n = k+1; n < N; n++)
                TASK_dtsmqr(quark,, A(k,n), A(m,n),
                             A(m,k), T(m,k));
        }
    }
    for k = 0 ... TILES-1
gqrt( A_{kk}^w, T_{kk}^w )
    for n = k+1..TILES-1
        unmqr( A_{kk-low}^r, T_{kk}^r, A_{kn}^w )
    for m = k+1..TILES-1
        tsqrt( A_{kk-up}^r, A_{mk}^r, T_{mk}^r )
        for n = k+1..TILES-1
            tsmqr( A_{mk}^r, T_{mk}^r, A_{kn}^w, A_{mn}^w )
```

Asim YarKhan & Jack Dongarra (UTK) Lightweight Superscalar Task Execution Nov 16 2014 10 / 24
Productivity: QUARK-D QR Implementation

The task is inserted into the runtime and held till data is ready.

```c
void TASK_dgeqrt(
    Quark *quark , . , int m, int n,
    double *A, int A_home, key *A_key ,
    double *T, int T_home, key *T_key )
{
    QUARKD_Insert_Task( quark , CORE_dgeqrt , . . ,
        VALUE, sizeof( int ) ,&m,
        VALUE, sizeof( int ) ,&n ,
        INOUT | LOCALITY , sizeof(A) ,A,A_home,A_key ,
        OUTPUT, sizeof(T) ,T,T_home,T_key , . , 0 ) ;
}
```

When the task is eventually executed, the dependencies are unpacked, and the serial core routine is called.

```c
void CORE_dgeqrt( Quark *quark )
{
    int m,n,ib,lda,ldt ;
    double *A,*T,*TAU,*WORK;
    quark_unpack_args_9( quark ,m,n,ib,A,
        lda ,T,ldt,TAU,WORK) ;
    CORE_dgeqrt(m,n,ib,A,lda,T,ldt,TAU,WORK) ;
}
```
Distributed Memory Algorithm

This pseudocode manages the distributed details for the user.

```
// running at each distributed node
for each task $T$ as it is inserted
  // determine $P_{exe}$ based on dependency to be kept local
  $P_{exe}$ = process that will run task $T$
  for each dependency $A_i$ in $T$
    if (I am $P_{exe}$) && (A$_i$ is invalid here)
      insert receive tasks ($A_i^{rw}$)
    else if (P$_{exe}$ has invalid $A_i$) && (I own $A_i$)
      insert send tasks ($A_i^r$)
  // track who is current owner, who has valid copies
  update dependency tracking
  if (I am $P_{exe}$)
    insert task $T$ into shared memory runtime
```
Execution of a small QR factorization (DGEQRF). Three processes (P0, P1, P2) are running the factorization on 3x3 tile matrix using a $1 \times 3$ process grid. Note that TSQRT and TSMQR have locality on second RW parameter.

Asim YarKhan & Jack Dongarra (UTK)  Lightweight Superscalar Task Execution  Nov 16 2014 13 / 24
QUARK-D’s principles of operation. Scheduling the DAG of the distributed memory QR factorization. Three distributed memory processes are running the factorization algorithm on a 3x3 tile matrix. One multi-threaded process runs all the blue tasks, another multi-threaded process runs the green tasks, and a third runs the purple tasks. Colored links show local task dependencies. Black arrows show inter-process communications.
QUARK-D: Key Developments

- **Distributed scheduling**
  - A function tells us which process is going to run a task; usually based on data distribution (2D block cyclic) but any function that will evaluate the same on all processes.
  - Execution within a multi-threaded process is completely dynamic.

- **Decentralized data coherency protocol**
  - Processes coordinate the data movement without any control messages.
  - Coordination is enabled by a data coherency protocol, where each process knows who is the current owner of a piece of data, and which processes have valid copies of that data.

- **Asynchronous data transfer**
  - Data movement is initiated by tasks, then the message passing continues asynchronously without blocking other tasks.
  - The data movement protocol is an eager protocol initiated by a send-data task. The receive-data task is activated by the message passing engine, and can get the data asynchronously (from temporary storage if necessary).
Figure: Trace of a QR factorization of a matrix consisting of 16x16 tiles on 4 (2x2) distributed memory nodes using 4 computational threads per node. An independent MPI communication thread is also maintained. Color coding: MPI (pink); GEQRT (green); TSMQR (yellow); TSQRT (cyan); UNMQR (red).
Figure: Weak scaling performance of QR factorization on a small cluster. Factorizing a matrix (5000x5000/per core) on up to 16 distributed memory nodes with 8 cores per node. Comparing QUARK-D, PaRSEC and ScaLAPACK (MKL).
Figure: Weak scaling performance for QR factorization (DGEQRF) of a matrix (5000x5000/per core) on 1200 cores (100 distributed memory nodes with 12 cores per node). Comparing QUARK-D, PaRSEC and ScaLAPACK (libSCI).
Figure: Weak scaling performance of Cholesky factorization (DPOTRF) of a matrix (5000x5000/per core) on 16 distributed memory nodes with 8 cores per node. Comparing QUARK-D, PaRSEC and ScaLAPACK (MKL).
Figure: Weak scaling performance for Cholesky factorization (DPOTRF) of a matrix (7000x7000/per core) on 1200 cores (100 distributed memory nodes with 12 cores per node). Comparing QUARK-D, PaRSEC and ScaLAPACK (libSCI).
DAG Composition: Cholesky Inversion

- Cholesky Inversion
- POTRF, TRTRI, LAUUM
- DAG composition can compress DAGs substantially
Figure: Trace of the distributed memory Cholesky inversion of a matrix with three DAGs that are composed (POTRF, TRTRI, LAUUM)
Designed and implemented a runtime system for task based applications on distributed memory architectures.

Uses serial task insertion interface with automatic data dependency inference.

No global coordination for task scheduling.

Distributed data coherency protocol manages copies of data.

Fast communication engine transfers data asynchronously.

Focus on *productivity, scalability* and *performance*. 
The End